Channel Routing With CLP(FD)

Denys Duchier
Serge Le Huitouze
Intelligent Software Group
Simon Fraser University
{duchier,serge}@cs.sfu.ca

1 Introduction

Channel routing is an important problem for the automated layout of integrated circuits. In [10], Neng-Fa Zhou claims that traditional CLP languages, such as CHIP, are ill-suited to the task, and that multi-layer channel routing problems are difficult to solve with finite domain variables ranging over integers. Instead, he describes a technique for explicit management of finite domains using β-PROLOG's state tables, and argues for its superior efficiency.

We set out to challenge that claim, and in the present paper we report on our experience applying finite domains to the channel routing problem using CLP(FD) [2, 3]. Our approach is quite elegant. We believe it is simpler than Zhou's β-PROLOG version, but, more to the point, it is just as performant.

We begin with a description of the channel routing problem and contribute a formalization which is well-suited to a constraint-based approach. Then we present Zhou's solution.

In section 5.1 we introduce a preliminary version (using two finite domain variables per net) which is the natural encoding of our formalization, but whose efficiency leaves something to be desired. Then, in section 5.2 we describe a better encoding (using only one finite domain variable per net) whose performance compares favorably with Zhou's solution.

In section 5.3, we document a useful technique for representing and updating graphs using finite domain variables. Finally, we report on and discuss comparative benchmarks: they validate our claim that CLP(FD) is competitive with β-PROLOG for this problem.

2 Problem Description[1]

The *channel* consists of a rectlinear grid of rows (aka *tracks*) and columns. The grid points along the top and bottom tracks are called *terminals*; they are

[1] This section is endebted to the excellent description of channel routing in [1].

numbered from 1 to n according to the column in which they lie. A *net* is a set of terminals which must be interconnected. A terminal can be in at most one net. By convention, we reject the degenerate cases of empty and singleton nets: a net must have at least 2 terminals.

How the problem can be solved further depends on the wiring model; in particular, it depends on the number of layers available. In the Manhattan routing model, 2 layers are available: all horizontal wires go in one layer and all vertical wires in the other; thus wires can cross, but they cannot overlap.

We also impose the "no dogleg" restriction. This means that the wiring of a net contains at most one horizontal segment to which all the net's vertical segments are connected.

Finally, we will allow k pairs of layers to be used as described above, so that a net may be placed on any one pair.

As a matter of convenience, we will say "layer" instead of "pair of layers," and the number of tracks will not include the top and bottom tracks because we make the further assumption that these two tracks can't be used for (horizontal) wiring.

3 Problem Formalization

A net will be implemented by a wiring consisting of one horizontal segment spanning the width from the net's letfmost terminal to its rightmost terminal, and one vertical segment per terminal connecting it to the horizontal segment. Our only constraint is that wires going in the same direction should not overlap. This can only happen when 2 nets' horizontal extents intersect. In that case, the horizontal segments cannot be laid on the same track: either they must be on different layers, or on the same layer but on different tracks. Furthermore, vertical segments should not overlap either: this can only happen when one net includes top terminal i and the other net includes bottom terminal i. To prevent overlap of these vertical segments, either the nets must be laid on different layers, or the net with top terminal i must be laid on a track above that of the net with bottom terminal i.

Note that, when the horizontal extents of two nets overlap but they do not share a column, then they must be laid on different tracks, but there is no ordering constraint between these tracks.

Let us write η for a net, $\eta.\mathbf{t}$ (resp. $\eta.\mathbf{b}$) for the set of indices of top (resp. bottom) terminals in net η. Let $\eta.\mathbf{e}$ be the extent of net η, i.e. the interval $[k_1, k_2]$ where k_1 is the minimum index of the terminals in η and k_2 is the maximum.

Different Tracks. When two nets η_1 and η_2 have overlapping extents, they must be placed on different tracks, which we write $\eta_1 \not\approx_T \eta_2$:

$$\eta_1.\mathbf{e} \cap \eta_2.\mathbf{e} \neq \emptyset \Rightarrow \eta_1 \not\approx_T \eta_2$$

Ordered Tracks. When two nets have terminals in the same column, either they must be placed on different layers or the net η_1 with the bottom terminal must be placed below the net η_2 with the top terminal. We write $\eta_1 \prec \eta_2$:

$$\eta_1.\mathbf{b} \cap \eta_2.\mathbf{t} \neq \emptyset \Rightarrow \eta_1 \prec \eta_2$$

Different Layers. When two nets must simultaneously satisfy incompatible ordering constraints, they must be placed on different layers, which we write $\eta_1 \not\approx_L \eta_2$:

$$\eta_1 \prec \eta_2 \wedge \eta_2 \prec \eta_1 \Rightarrow \eta_1 \not\approx_L \eta_2$$

A solution Γ in n layers and m tracks is a set of elements $\eta : \langle i, j \rangle$ assigning a layer number $1 \leq i \leq n$ and a track number $1 \leq j \leq m$ to each net, and such that it respects all constraints on non-overlap:

Different Tracks. Γ satisfies $\eta_1 \not\approx_T \eta_2$ iff $\eta_1 : \langle i_1, j_1 \rangle \in \Gamma$ and $\eta_2 : \langle i_2, j_2 \rangle \in \Gamma$ and $i_1 \neq i_2$ or $j_1 \neq j_2$.

Ordered Tracks. Γ satisfies $\eta_2 \prec \eta_1$ iff $i_1 \neq i_2$ or $j_2 < j_1$.

Different Layers. Γ satisfies $\eta_1 \not\approx_L \eta_2$ iff $i_1 \neq i_2$.

4 β-Prolog Solution

Zhou presents a solution that takes advantage of β-prolog's support for *state tables*, i.e. "relations in which each tuple is given a truth value *true* or *false*." He treats the problem as a CSP where the domains of variables are represented by a state table whose triples are all combinations of nets, layers and tracks. A state table can also be regarded as a multidimensional array of boolean values indexed by atomic terms.

Creation. The goal `bt(p(X1,...,Xn),S)` creates a state table `p` with n dimensions, where each dimension is specified by the range `Xi`, i.e. a list of atoms denoting the legal indices on the ith dimension, and `S` is a boolean indicating the initial state of the entries. A range of integers can also be written $i..j$.

For example `bt(p1([a,b,c],1..4),true)` creates a 3×4 array, with each entry initially `true`. The first dimension is accessed by either `a`, `b` or `c`, and the second one by either `1`, `2`, `3` or `4`.

Modification. A state table can be modified using primitives `set_true` and `set_false`. For example, `set_false(p1(a,2))` will set entry `p1(a,2)` to `false`. These primitives also accept *tuple patterns* as arguments, i.e. tuples where the first few arguments are atomic values (or intervals) and the remaining arguments are distinct variables; all entries matching the pattern will be modified. If a state table is regarded as a discriminating tree, invoking `set_false` prunes an entire subtree. Modifications are undone on backtracking.

Access. The goal `select(p1(a,X))` will non-deterministically enumerate upon backtracking all possible values of `X` for which table entry `p1(a,X)` is true. The goal `count(p1(a,_),C)` will instantiate `C` to the number of tuples `p1(a,X)` which are true in table `p1` (3 in our example).

Zhou's solution uses a state table `domain(Net,Layer,Track)` to represent the possible assignments that are still valid for the remaining nets. He proceeds with a traditional, heuristically guided, enumeration of the nets together with the propagation of constraints (here the update of the state table `domain`) after each instantiation. Thus propagation performs forward checking by updating the domains of the remaining variables.

Once a net `N` has been chosen, enumeration of its domain is simply achieved by the goal `select(domain(N,L,T))`. Corresponding updates must maintain the non-overlap constraints. Suppose we have just selected `domain(Ni,Li,Ti)`:

Different Tracks. If `Ni` and `Nj` must be on different tracks, we simply execute `set_false(domain(Nj,Li,Ti))`, thus eliminating the pair ⟨`Li`,`Ti`⟩ from the domain of `Nj`.

Ordered Tracks. If `Ni` must be above `Nj`, we correspondingly forbid tracks `Ti` and above on layer `Li` for `Nj` by executing `set_false(domain(Nj,Li,Ti..M))` where `M` is the number of tracks. If `Ni` must be below `Nj`, we eliminate tracks `1` to `Ti` instead with `set_false(domain(Nj,Li,1..Ti))`.

Different Layers. If `Ni` and `Nj` must be on different layers, we simply remove all entries on layer `Li` from the domain of `Nj`: `set_false(domain(Nj,Li,_))`.

Zhou represents the set of non-overlap constraints by two graphs, also implemented by state tables: a directed *vertical constraint graph* `Gv` capturing the "ordered tracks" constraint, and an undirected *horizontal constraint graph* `Gh` capturing the "different tracks" constraint. There is an edge from η_1 to η_2 in `Gv` iff $\eta_2 \prec \eta_1$. There is an edge between η_1 and η_2 in `Gh` iff $\eta_1 \not\approx_T \eta_2$. To avoid unnecessary propagations, these graphs are also updated: each time a net is selected, corresponding constraints are looked up in these graphs and then deleted.

Some heuristics are based on the in/out degrees of nodes in `Gv` and `Gh`, thus measuring the number of constraints in which a particular node is involved. In Zhou's solution, degrees are easily computed by the `count` primitive. The depth of nodes in `Gv` is also used: Zhou's implementation requires that there be no cycles in the graph; ours lifts this limitation by collapsing cycles. In both implementations, a node's depth is computed statically in the original constraint graph `Gv` and is not updated to reflect the evolving topology when nodes are deleted from `Gv` during labeling.

5 CLP(FD) Solution

CLP(FD) [2, 3] is a Constraint Logic Programming language where constraints apply to finite domain variables ranging over integers. The basic constraint is `X in R`, stating that the (finite domain) variable `X` must always be compatible with the range `R`, which means that the possible values that `X` can take must belong to the set represented by `R`.

A range can be a constant range (e.g. `1..10`). It can also be what is called an *indexical range*, involving the values and/or limits of other finite domain variables. For example, the range `min(Y)..max(Z)` represents the set of all integers between the minimum value in the domain of `Y` and the maximum value in the domain of `Z`.

A important requirement on ranges is that they must be negatively monotonic, that is, additional constraints can only make the domains smaller. It follows that ranges like `max(Y)..min(Z)` are forbidden: constraining variable `Y` would eventually decrease its possible maximum value, thus increasing the above range (the same reasoning applies to `Z` too in this example).

Other indexical ranges are `dom(V)` denoting the domain of a finite domain variable, and `val(V)`, denoting the value of a finite domain variable; the latter is only meaningful when `V` becomes instantiated, thus a constraint involving `val(V)` only takes effect when the domain of `V` has been reduced to a single integer.

Indexical ranges are very important in CLP(FD): they determine how constraints propagate whenever ranges change.

A little example might provide some insight on how this works:

```
rel1(X,Y) :-
  X in min(Y)+3..max(Y)+3,
  Y in min(X)-3..max(X)-3.

rel2(X,Y) :-
  X in dom(Y)+3,
  Y in dom(X)-3.

rel3(X,Y) :-
  X in {val(Y)+3},
  Y in {val(X)-3}.
```

These three predicates almost state the same constraint, namely `X = Y+3`. They only differ in the amount of propagation which is performed when the domains of `X` and `Y` are updated.

Suppose `X` and `Y` have current domain `1..10`. Then execution of `rel1(X,Y)` constrains the domain of `X` to `4..10` and the domain of `Y` to `1..7`.

Further stating `X in -{5}`[2] results in `X`'s domain being reduced to `{4}:6..10`, but no modification of `Y`'s domain (minimum and maximum values for `X`, namely

[2] Range $-r$ denotes the complement of range r.

4 and 10 have not been changed).

On the other hand, using rel2 instead would result in Y's domain being reduced to {1}:3..7.

Finally, using rel3 instead, X's and Y's initial domain would remain unchanged (1..10) after execution of rel3(X,Y) and the subsequent constraint X in -{5} would not affect Y's domain. Propagation would only take place when X (resp. Y) is instantiated to a particular integer, leading to instantiating Y (resp. X) to integer value X-3 (resp. Y+3).

To summarize, indexical constraints implement the following propagations:

- forward checking, with val(V),
- full lookahead, with dom(V),
- partial lookahead, with min(V) and max(V).

Moreover, it is possible to mix different propagation schemes in a single range: X in min(Y)..{val(Z)} performs partial lookahead on Y and forward checking on Z (this example is for illustration purposes only and should not be confused with good programming style!).

5.1 Solution with two variables per net

CLP(FD) is very well suited for solving finite domain CSP problems, provided you can express your constraints in terms of indexical ranges. If so, you can simply post all your constraints and then invoke an appropriate *labeling* procedure. As we shall see, the channel routing problem can be easily coded using finite domain constraints.

Our first solution is an almost direct encoding of the mathematical formulation: let's assume each net Ni is represented by two finite domain variables Li and Ti, denoting its layer and track number.

Different Layers. This constraint can be simply enforced by the library call 'x<>y'/2, which is implemented by forward checkable constraints:

```
different_layers(L1,T1,L2,T2,NbT) :- 'x<>y'(L1,L2).

'x<>y'(X,Y) :- X in -{val(Y)}, Y in -{val(X)}.
```

The effect will be to suppress X's (resp. Y's) value from Y's (resp. X's) domain as soon as it is instantiated to a particular integer.

The two other constraints are a little bit trickier because they involve a disjunction (recall the mathematical definition): either the two layers are different or some relation holds between the two tracks. It is well known that some disjunctions can be represented by addition. We tried to apply this idea to our problem.

Different Tracks. If T_i and T_j are the finite domain variables representing the tracks in question, the idea is to enforce $T_j + \delta_{ij} \neq T_i$ where δ_{ij} is 0 when the layers are equal and is a large number (bigger than the number of possible tracks) when the tracks are different (thus causing the constraint to be necessarily satisfied).

The challenge is to express δ_{ij} in terms of indexical ranges: we used the absolute difference between the layers, scaled by the maximum number of tracks. Also, instead of just computing δ_{ij}, we compute $T_j + \delta_{ij}$. We used a definition similar to the `'|x-y|=z'/3` library call. The idea is to consider two cases: either `x` is smaller than `y` or the other way around. Depending on the case, we are interested in either `y-x` or `x-y`. Merging these two intervals (by using `:`, the union constructor) will do the job, because eventually the 'wrong' one (i.e. the one yielding an interval of negative numbers) will be discarded due to the fact that CLP(FD) only manipulates non-negative integers.

This leads to the following definition, where `L1` and `L2` are the two layers, `T` is one of the tracks, `NbT` is the maximum number of tracks for the problem, and `PseudoT` is the resulting pseudo-track to be confronted with the other track:

```
pseudo_track_number(L1,L2,T,NbT, PseudoT) :-
  Delta in NbT*(min(L1)-max(L2) .. NbT*(max(L1)-min(L2)
         : NbT*(min(L2)-max(L1) .. NbT*(max(L2)-min(L1),
  T in min(PseudoT)-max(Delta) .. max(PseudoT)-min(Delta),
  PseudoT in min(Delta)+min(T) .. max(Delta)+max(T).
```

Now, we can express the constraint:

```
different_tracks(L1,T1,L2,T2,NbT) :-
  pseudo_track_number(L1,L2,T2,NbT, PseudoT2),
  'x<>y'(PseudoT2,T1).
```

Ordered Tracks. Here again, either the nets are on different layers or the first one must be placed below the second one. By enforcing $T_j + \delta_{ij} > T_i$ we are guaranteed to succeed when the layers are different (since δ_{ij} is so large), and we are enforcing $T_j > T_i$ when the layers are equal.

```
ordered_tracks(L1,T1,L2,T2,NbT) :-
  pseudo_track_number(L1,L2,T2,NbT, PseudoT2),
  'x>y'(PseudoT2,T1).
```

5.2 Solution with one variable per net

Our second solution is less intuitive, but shows the power of the glass-box approach used in CLP(FD).

Here, each net is represented by a single finite domain variable `LTi`, which is indeed a composite value of both the layer and the track number. More precisely, this variable corresponds to `Li*NbT+Ti`, assuming `Li` (resp. `Ti`) ranges over `[0..NbLayers-1]` (resp. `[0..NbT-1]`).

Different Tracks. We can simply enforce this constraint by the forward checkable constraint `'x<>y'/2`, because we forbid a particular pair of layer and track, i.e. a particular value for a composite variable.

```
different_tracks(LT1,LT2,NbT) :- 'x<>y'(LT1,LT2).
```

The two other constraints are also implemented in a forward checkable manner, by deleting an interval of values corresponding to a particular layer. All the difficulty here is to express this interval.

Given a particular value for `LTi`, the value for `Li` is simply `LTi//NbT` (`//` stands for the integer division). The value for `Ti` can be computed accordingly.

Different Layers. Suppose nets `Ni` and `Nj` must be laid on different layers. When finite domain variable `LTi` is instantiated to a particular value, one has to forbid values corresponding to `Li` for `LTj`, that is delete the interval `[Li*NbT..Li*NbT+NbT-1]`.

```
different_layers(LT1,LT2, NbT) :-
  LT1 in -( (val(LT2)//NbT)*NbT .. (val(LT2)//NbT)*NbT+(NbT-1) ),
  LT2 in -( (val(LT1)//NbT)*NbT .. (val(LT1)//NbT)*NbT+(NbT-1) ).
```

Ordered Tracks. If net `Ni` must be placed below net `Nj`: whenever `LTi` is instantiated to a particular value corresponding to `Li` and `Ti`, we must remove all values from `Li*NbT+0` to `Li*Nbt+Ti` from the domain of `Nj`; whenever `LTj` is instantiated with a value corresponding to `Lj` and `Tj`, we must remove all values from `Lj*NbT+Tj` to `Lj*NbT+NbT-1` from the domain of `Ni`.

```
ordered_tracks(LT1,LT2,NbT) :-
  LT1 in -( val(LT2) .. (val(LT2)//NbT)*NbT+(NbT-1) ),
  LT2 in -( (val(LT1)//NbT)*NbT .. val(LT1) ).
```

5.3 Another application of finite domain variables

Certain heuristics employed during the labeling process make use of the in/out degrees of the horizontal and vertical constraint graphs. We chose not to represent these graphs explicitly. Instead, we use a finite domain variable for each node and each graph. The domain of these variables represent the nodes that are connected to this particular node (i.e. net) in this particular graph. Hence, the degree can be computed simply by the builtin predicate `fd_size/2`.

Some caution must be taken, however, in the case where a node becomes completely disconnected from the rest of the graph. In such a case, according to our previous definition, the associated finite domain variable would have an empty domain, thus causing a failure! To circumvent this problem, we had to extend the domains of these variables with an extra *fake* value: since we numbered nets from 1 to n, we chose 0 for this extra value.

As an example, consider a vertical constraints graph consisting of four nodes and three arcs : $\{(1,2),(2,3),(2,4)\}$. The initial domains for the variables representing degrees for the node 2 are `{0}:{1}` for the in-degree, `{0}:{3}:{4}` for the out-degree.

The update amounts to deleting the chosen net (in the labeling process) from all degree variables. This can be achieved simply by constraining all the degree variables to be included in the domain of an extra finite domain variable representing the set of the nodes still to be instantiated. So, for every degree variable `DVi`, we must install the following constraint: `DVi in dom(NonInstNodes)`. The propagation implemented by `dom` being a full lookahead, each modification of `NonInstNodes` in the labeling process is immediately reflected in the domains of the degree variables, thus insuring that correct values are available to the heuristics.

This is not a canonical use of finite domain variables: degree variables are not part of any labeling process. Their possible values do not correspond to values that can be enumerated for them. They are just, we think, a convenient way of representing the degrees of particular graphs, and perhaps the only reasonable one in a language lacking backtrackable assignment.

Our trick works fine on this example because we start with initial graphs (representing the set of all constraints) and we suppress nodes (i.e. nets) from the graphs as execution proceeds. If non-monotonic modifications were required, we could not get away with it. Note, however, that if you need to add rather than suppress values, you could operate on the complement of your domain instead.

6 Results

We tried our two different versions against Zhou's version with different heuristics. After some experimentation, we found that the best heuristic ordering for net selection was as follows:

- first we select nets with the fewest number of ordering constraints placing nets below them: i.e. with minimum out-degree in `Gv`. Since values are enumerated in increasing order, starting from the low end of a domain, this policy minimizes the likelihood of picking a value that will subsequently prove incompatible with a net to be placed below.
- then, the nets which are deepest in `Gv`, i.e. which have constraining effect on the largest number of other nets above them, directly or indirectly. This maximizes the pruning effect of solving for this net.
- then, the nets with the largest in-degree in `Gv`, i.e. nets which have the most direct effect on nets above them.
- then, the nets with the largest degree in `Gh`, i.e. nets with the most number of difference contraints.

In the sequel, we refer to this heuristics as H1.

The measurements were made on Deutsch's difficult problem [5], for various numbers of layers with the corresponding minimal number of tracks, namely 1 layer/28 tracks, 2 layers/11 tracks or 3 layers/7 tracks.

The measurements reflect only the time required for finding the first solution, excluding the initialization phase. The raw results are in milliseconds and were obtained on a Sparc 10, 40 MHz. Also included in the tables is a corrected ratio between the two versions, taking into account the 1.76 average speed-up of β-prolog over CLP(FD) which we measured on a few benchmark programs — our purpose was to evaluate the use of state tables vs. finite domain variables, not to compare the raw speed of β-prolog's implementation against that of CLP(FD)'s.

For this problem, the initialization time was about 1800 msec for the CLP(FD) version, and 1000 msec for the β-prolog version. Considering the speed-up factor, these times are almost the same.

6.1 Version with two variables per net

	BetaProlog	clp(FD)	ratio
1 layer	253	642	1.44
2 layers	250	728	1.65
3 layers	250	777	1.77

This version is quite efficient, though not as efficient as Zhou's on this benchmark. However, its performance decreases markedly with the number of layers.

Zhou's version performs only forward checking, ours performs partial lookahead in the 1 layer case for the "Ordered Tracks" constraint. However, lookahead isn't helpful in this particular case.

In the multi-layer case, our version doesn't perform immediate forward checking. Due to the coding of the two disjunctive constraints (see `pseudo_track_number`, p7), ordering constraints on track values of two nets will normally only take effect after instantiation of the layer values. This might delay failures detected earlier in Zhou's version, and that is probably the cause of the slight degradation one can observe in the measurements.

However, with this particular heuristics, this version still performs quite well.

We experimented with two other heuristics also used in Zhou's program. Though not useful for solving this example more efficiently (even in Zhou's version), they showed a marked penalty in our version.

H2 uses, in addition to H1, the minimum values `(L,T)` for nets: we choose first, all other things being equal, the net with the smaller minimum. This cannot be simply implemented as the minimum value of `L` together with the minimum value of `T`, because of the delayed forward checking our version implements. Thus, we have to enumerate on the layer value until a proper value is found. Only then can we pick up the minimum value for the track. Of course,

we must then undo the bindings and propagation just performed. The code looks as follows:

```
min_val(Layer,Track, _,_) :- first_val(Layer,Track), fail.
min_val(_,_, MinL,MinT) :- retract(minv(MinL,MinT)).

first_val(Layer,Track) :-
  in_domain(Layer), fd_min(Track,MinT), !,
  asserta(minv(Layer,MinT)).
```

	BetaProlog	clp(FD)	ratio
1 layer	292	1060	2.06
2 layers	299	6865	13.05
3 layers	303	8530	16.00

H3 uses, in addition to H1, the number of remaining values `(L,T)` for nets: we choose first, all other things being equal, the net with the smaller number of remaining values. The implementation for this measure is even worse than the previous one! We have to perform a complete enumeration on the layer value and sum the numbers of possible values for the track. The code looks as follows:

```
size_dom(Layer,Track, _) :-
  asserta(sz(0)),
  indomain(Layer), fd_size(Track, NbTracks),
  update_sz(NbTracks),
  fail.
size_dom(_,_, Size) :- retract(sz(Size)).

update_sz(NbTracks) :-
  retract(sz(S0)), S1 is S0+NbTracks, asserta(sz(S1)).
```

	BetaProlog	clp(FD)	ratio
1 layer	278	1248	2.17
2 layers	285	10171	20.28
3 layers	285	14885	29.68

Again, it is important to note that these heuristics are not necessary. Even in Zhou's version, H1 is more efficient. In our "two variables per net" design, they are very expensive to compute; however, in our "one variable per net" design they become quite cheap (see below).

6.2 Version with one variable per net

Heuristic H1

	BetaProlog	clp(FD)	ratio
1 layer	253	435	0.98
2 layers	250	440	1.00
3 layers	250	435	0.99

As opposed to the previous one, this version performs exactly the same pruning at exactly the same time as Zhou's version. The efficiency is the same too. Furthermore, heuristics H2 and H3 can be coded trivially using `fd_min` and `fd_size` respectively.

Heuristic H2

	BetaProlog	clp(FD)	ratio
1 layer	292	476	0.93
2 layers	299	467	0.89
3 layers	303	462	0.87

Heuristic H3

	BetaProlog	clp(FD)	ratio
1 layer	278	469	0.96
2 layers	285	469	0.93
3 layers	285	461	0.92

As can be seen, our version is even a little quicker than Zhou's with H2 and H3.

7 Conclusion

We presented a formalization of the channel routing problem which is well-suited to a CLP approach: it dictates the constraints which must be posted a priori before proceeding with a labeling phase.

We then described a concrete implementation of multi-layer dog-leg free channel routing in CLP(FD) using finite domain variables. We also contributed a useful technique for representing and updating graphs using finite domain variables.

Finally we reported our comparative measurements on Deutsch's difficult problem [5]. They disprove Zhou's conjecture [10] that finite domain techniques are inappropriate for solving channel routing problems: our solution is not only elegantly declarative, it is also as performant as Zhou's β-PROLOG version.

We are grateful to Neng-Fa Zhou for bringing the problem to our attention and for stimulating discussions during this research.

References

[1] Bonnie Berger, Martin Brady, Donna Brown, Tom Leighton. *Nearly Optimal Algorithms and Bounds for Multilayer Channel Routing.* Journal of the ACM, v42 n2, March 1995, pp 500–542.

[2] P. Codognet and D. Diaz. *A Minimal Extension of the WAM for* CLP(FD). In *10th International Conference on Logic Programming*, Budapest, Hungary, MIT Press, 1993.

[3] P. Codognet and D. Diaz. *Compiling Constraint in* CLP(FD). To appear in *Journal of Logic Programming.*

[4] D. Diaz. CLP(FD) *2.21 User's Manual.* INRIA, July 1994. `ftp://ftp.inria.fr/INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/clp_fd`

[5] David N. Deutsch. *A "Dogleg" Channel Router.* Design Automation Conference, 1976.

[6] Sung-Chuan Fang, Wu-Shiung Feng and Shian-Lang Lee. *A New Efficient Approach To Multilayer Channel Routing Problem.* 29th ACM IEEE Design Automation Conference, 1992.

[7] Shaodi Gao and Michael Kaufmann. *Channel Routing of Multiterminal Nets.* Journal of the ACM, v41 n4, July 1994, pp. 791–818.

[8] Xingzhao Liu, Akio Sakamoto and Takashi Shimamoto. *Genetic Channel Router.* 6th Karuizawa Workshop on Circuits and Systems, IEICE Trans. Fundamentals, vE77–A n3, March 1994.

[9] H. Simonis. *Channel Routing Seen as a Constraint Problem.* ECRC TR-LP-51, 1990.

[10] Neng-Fa Zhou. *A logic Programming Approach to Channel Routing.* Kyushu Institute of Technology, Japan. ICLP'95.